Data Palooza

What is a variable?

A variable is a symbolic name associated with a storage location that contains a value or a pointer (to a value)

What is a value?

A value is a piece of data with a type, this is either referred to by a variable or computed by a program expression.

What are all facets that make up a variable?

1. Name: how you refer to the variable
2. type: a variable may (or may not) have an assigned type
3. value: the value being stored and its type
4. Binding - how a variable name is connected to its current value
5. storage: the memory slots that holds the value
6. lifetime: the timeframe over which a variable exists
7. scope: when/where the variable name is visible to code
8. mutability: can a variable's value be changed

Higher RAM Addresses Stack - local variables, parameters Heap - Dynamically allocated objects Static data - global variables, static variables

Types, Scoping and Lifetime, Memory Safety, Mutability, binding semantics

In python/dynamically typed language, much more about the value rather than the actual type Mutability: Immutability might seem like it makes the language less flexible (and it does to some extent), it also allows for the program to be less buggy, and can also allow for some compiler optimisations to take place.

What is type?

A type is a classification that is used to identify a category of data. A variable's/value's type influences its size and encoding, what operations we can perform on it, where it can be used, and how it's converted/cast to other types.

In a typed language, not every variable must have a type. if a given variable is "bound" to a single value, then it can be said to have a type otherwise not

A value is always associated with a type

Types of types

Primitives: Integers, Floating Point Numbers, characters, Enumerated types (ordinals), booleans, pointers Composites: Records (structs), unions, classes, strings, tuples, containers (Arrays, Sets, lists, maps...) Others: Generic types, function types, boxed types A boxed type is just an object whose only data member is a primitive (like an int or a float)

User-defined types

Beyond built-in types like int, double, string... languages also let users define new types

- using classes or structs, or enum, or interface in JavaScript
- An interface is a list of function declarations - it's like a fully-abstract class with no implementations or fields

Value Types and Reference Types

Value Types A value type is one that can be used to instantiate objects/values (and define pointers/object references/references)

```
class Dog:
public:
    Dog(string n) { name_ = n; }
    void bark() { cout << "Woof\n"; }
private:
    string name_;
Dog d("kuma"), *p;
```

A reference type can only be used to define pointers/object references/references but not instantiate objects/values.

```
class Shape:
public:
    Shape(Color c) { color_ = c; }
    virtual double area() = 0; // the abstract method of pure virtual is just a prototype
private:
    Color color_;
Shape s(Blue); // won't work
// It's reference type because we can't use the type to instantiate objects because the class is missing at least one implementation
// But we can use the type to define pointers/object references
```

Mojo:

- a superset of python. A compiled language that combines the simplicity of Python with the speed of C++ There are no built-in types (even int, string, etc)! Uses "autotuning" to optimize a program based on actual execution patterns and hardware configurations.

Type Checking Approaches

Strong type checking - language's type system guarantees that all operations are only invoked on objects/values of appropriate types Weak type checking - language's type system does not guarantee that all operations are invoked on objects/values of appropriate types

Compile-time vs run-time

Static typing: prior to execution the type checker determines the type of every expression and ensures all operations are compatible with the types of their operands Dynamic typing: as the program executes, the type checker ensures that each primitive is invoked with values of the right types, and raises an exception otherwise

Examples of Static, Strong C#, Go, Haskell, Java, Scala

Examples of Dynamic, Strong Javascript, Perl, PHP, Ruby, Python

Examples of Static, Weak Assembly Language, C, C++

Examples of Dynamic, Weak None???

Static Typing

a typechecker checks that all operations are consistent with the types of the operands being operated on prior to the program's execution If the type checker can't assign distinct types to all variables, functions and expressions and verify type compatibility, then it generates a compiler error.
But if program type checks, it means the code is largely type-safe and few if any checks need to be done at runtime.

To support static typing, a language must have a fixed type bound to each variable at its time of definition. Once a variable's type is assigned, it can't be changed. Consider C++ (statically typed) and Python (dynamically typed) For the C++ code: the type of variable a is fixed at its definition and can't change For python, there's no way that a compiler could figure out the type of variable a when compiling this code.

Type Inference with Static Typing

Types don't need to be explicitly annotated for static typing: types can be often be inferred e.g. Haskell

In C++, the auto keyword can be used to infer the variable's type from the right-hand side expression In Java, the var keyword can infer the type of variables

Static Type Checking is Conservative Static type Checking can prevent techincally correct programs from compiling Because in order to guarantee type safety the type checker must be overly conservative.

Pros and Cons of Static Type Checking

Pros: Produces Faster code (no type checks at runtime, optimizations possible) Detect bugs earlier in development No need to write custom code to check types

Cons: Static Type checking is conservative and may error-out on perfectly valid code Static Typing requires a type checking phase before execution, which can slow development

Dynamic Typing:

In a dynamically typed language, the safety of operations on variables/values is checked as the program runs (rather than compile time) If the code is attempting an illegal operation on a value, an exception is generated or the program crashes.

Dynamic Typing: Types Associated with values! In dynamically-typed languages, variables don't have types. So a given variable name could refer to multiple values of different types over the course of execution.

TYPES ARE ASSOCIATED WITH VALUES AND NOT VARIABLES

How does a program written in a dynamically typed language detect type violations at runtime? The compiler uses type tags - secretly stored along w/ the value Type tags

are an extra piece of data stored along with each value/object that indicates what type it is

Dynamic Type Checking in Statically typed languages? Sometimes we need dynamic type checking in statically typed languages: Specifically for down-casting

```
class Person {...};
class Student : public Person { ... };
class Doctor : public Person { ... };
void partay(Person *p) {
    Student *s = dynamic_cast<Student *p> (p);
    if (s != nullptr)
        s -> getDrunkAtParty();
}
```

When we do the downcast, C++ checks in real-time whether the object passed in is compatible with the downcast. Dynamic Type Checking Pros and Cons: What are the pros of dynamic type checking?

- Increased flexibility
- It's often easier to implement generics that operate on many different data types
- Simpler code due to fewer type annotations
- Makes for faster prototyping

What are the cons of dynamic type checking?

- We detect errors much later
- code runs slower due to run-time type checking
- Requires more testing for the same level of assurance
- No way to guarantee safety across all possible executions

A weird consequence of Duck Typing: DuckTyping

With static typing, the compiler/interpreter can determine whether an operation will work on a variable based on its type. And it can check this statically before the program runs! But in dynamically-typed languages, variables have no type

```
# Python "duck" typing
class Duck:
 def swim(self):
  print("Paddle paddle paddle")

class OlympicSwimmer:
 def swim(self):
  print("Back stroke back stroke")

class Professor:
 def profess(self):
  print("Blah blah blah blah blah")

def process(x):
 x.swim()

def main():
```

```
d = Duck()
s = OlympicSwimmer()
p = Professor()
process(d) # Paddle paddle paddle
process(s) # Back stroke back stroke
process(p) # throws AttributeError
```

So they have no choice but to wait until an operation runs to check if it's valid. This means that duck typing is the only option in dynamically typed languages

Duck Typing: Python

Supporting Enumeration If you add **iter** and **next** methods to any class, you can make it "iterable!" Making any Class Printable If you add the **str** method to any class, you can make it "printable!" Testing for Equality of Value If you add the **eq** method to any class, you can make its objects "comparable"

Gradual Typing

Some variables may be given explicit types, others may be left untyped. Type checking occurs partly before execution and partly during runtime.

- you can choose whether to specify a type for variables/parameters
- If a variable is untyped, then type errors for that variable are detected at runtime
- But if you do a specify a type, then some type errors can be detected at compile time
- If we pass an untyped variable to a typed variable, it will compile fine
- Since you could pass an invalid type, the program will check for errors at runtime
- In gradual typing, some variables may have explicitly annotated types, while others may not. This allows the type checker to partially verify type safety prior to execution, and perform the rest of the checks during run time.
- With gradual typing, you can choose whether to specify a type for variables/parameters. If a variable is untyped, then type errors for that variable are detected at run time.

Strongly Typed Language

- Strongly-typed language ensures that we will never have undefined behavior at run time due to type-related issues.
- there's no possibility of an unchecked runtime type error.
  TO BE STRONGLY TYPED
  - the language must be type safe:
    - prevent an operation on a variable X if X's type doesn't support that operation
  - language must be memory-safe:
    - prevents inappropriate memory access (e.g. out-of-bound array accesses, access to a dangling pointer) These can be statically or dynamically enforced

Things we expect in a strongly typed language

- Before an expression is evaluated, the compiler/interpreter validates that all of the operands used in the expression have compatible types.

- All conversions/casts between different types are checked and if the types are incompatible (e.g. convverting an int to a Dog), then an exception will be generated.
- Pointers are either set to nill or assigned to point at a valid object at creation.
- Accesses to arrays are bounds checked; pointer arithmetic is bounds-checked
- The language ensures objects can't be used after they are destroyed.

General principle: Prevent operations on incompatible types or invalid memory

Memory Safety and Strong Typing

If a language is not memory safe, you might access a value using the wrong type A checked cast is a type-cast that results in an exception/error if the cast is illegal! Dramatically-reduced software vulnerabilities (less hacking) Earlier detection and fixing of bugs/errors

People still use weakly typed languages Performance and legacy

Weakly Typed language

They are not Type-safe: the language may not detect or prevent operations on data types that don't support those operations. Thet are not memory-safe: Programs may access memory outside of array bounds or via dangling pointers. Weakly types languages can have undefined behavior at runtime

Type Conversions, Type Casts and Coercions

Type conversion, typecasting, and coercion are different ways of explicitly or implicitly changing a value of one data type into another.

Super-Types and Sub-types

Given two types, Tsuper and Tsub, we say: Tsub is a subtype of Tsuper iff Requirement #1: Every element belonging to the set of values of type Tsub is also a member of the set of values of type Tsuper Requirement #2: All operations (e.g. +, -, *, /) that you can use on a value of type Tsuper must also work properly on a value of type Tsub.

If we have cpode designed to operate on a value of type Tsupe, it must also work if I pass in a value of type Tsub. Ex. Short is subtype of Int Ex. Float is not a subtype of Int because float can't represent everything an int can Ex. ints and const ints represent the exact same set of values.

When you use subclass inheritance... The base class definition implicitly defines a new type and the subclass definition implicity defines a new type too

Type Conversions and Type Casts

Type conversion and type casting are used when we want to perform an operation on a value of type A, but the operation requires a value of type B

- we want to pass an int value to a function that accepts a float value
- we want to add a long value to a double value in an expression
- we want to pass a Student object to a function that accepts a Person object (assuming Student is derived from Person)

Two Options

Type Conversion:

A conversion takes a value of Type A and generates a whole new value (occupying new storage, with a different bit encoding) of type B. Type conversions are typically used to convert between primitives (float -> int)

```
int main() {
    float pi = 3.14;
    cout << (int) pi
}
```

The program performs a conversion, and generates a temporary new value of a different type in the process. The converted value occupies distinct storage and has a different bit representation than the original value.

Type Casting

A cast takes a value of type A and views it as if it were value of type B - no conversion takes place! Type casts are typically used with objects

```
int main() {
    int val = -42;

    cout << (unsigned int) val;
    // prints 4294967254
}
```

This cast lets us refer to our original Student object, but interpret it as if it were just a Person This refers to our original integer, but "interprets" its bits as if they represented an unsigned int

Casts and Conversions: Three Categories

explicit vs. implicit widening vs. narrowing checked vs. unchecked

Explicit vs. Implicit Both conversions and casts can be explicit or implicit

An explicit conversion / explicit cast requires you to use explicit syntax to force the conversion/cast

```
Explicit conversion
void foo(int i) {...}
int main() {
    float f = 3.14;
    foo((int) f);
}
```

```
// Explicit Cast
void feed_young(Animal *a) {
    if (a->has_fur()) {
        ((Mammal *)a)-> produce_milk();
    }
}
```

An implicit conversion/ implicit cast is one which happens without explicit syntax

```
Implicit Conversion
void foo(float f) {...}
int main() {
    int i = 42;
    foo(i);
}
```

```
// Implicit Cast
void use_potty(Person *p) { p -> poop(); }
int main() {
    Nerd *n = new Nerd("paul");
    use_potty(n);
}
```

Most implicit casts are "upcasts" - from a subclass to a superclass Here we are implicitly upcast a Nerd object to a Person

Why do we have explicit conversions and casts? When you use an explicit conversion or cast, you're telling the compiler to change what would be a compile time error into a runtime check.

Implicit conversions: Coercions and Promotions

Most languages have a prioritized set of rules that govern implicit conversions (aka coercions) that are allowed to occur without warnings/errors.

Conversions: Widening vs. Narrowing

A widening conversion is one that converts a subtype value to one of its supertypes, e.g. int -> long, float -> double

```
void foo(int i) { ... }
int main() {
    short s = 42;
    foo(s);
}
```

Since a supertype can represent every value the subtype can, widening conversions are "value-preserving" - the converted value is always the same as the original

A narrowing conversion is one that converts from supertype to a subtype, or between two unrelated types

```
// Narrowing: int -> short
int main() {
    int i = 40000;
    cout << static_cast<short>(i);
}
```

Narrowing conversions are NOT value-preserving, meaning the converted/casted value might be different than the original

Casts: Widening vs. Narrowing

Casts can also be known as "upcast" or "downcast" A widening cast, aka an "upcast", casts a subtype variable as its supertype, e.g.: Student -> Person

```
class Person {...}
class Student: public Person {...}
void chat_with(Person &p) {
    cout << "Hi" << p->get_name();
}
int main() {
    Student s("Tammy", "CS");
    chat_with(s);
}
```

Upcasts are always safe because we know that every subtype object (e.g., Student) is guaranteed to have all of the properties of the supertype (e.g. Person)

Narrrowing cast, aka "downcast", is one that casts supertype variable as its one of its subtype, e.g.: Person -> Prof

```
class Person {...}
class Student: public Person {...}
void do_thing(Person *p) {
    if (p->get_name() == "Carey") {
        Prof *q = dynamic_cast<Prof *>(p);
        q->give_lecture();
    }
    else p->talk();
}
```

Downcasts may fail if the actual object is not compatible with the downcasted type

Conversions/Casts: Checked or Unchecked

Conversions and Casts can be checked or unchecked. In a strongly-typed language, every conversion/cast with the potential for an issue is checked for validity at runtime.

In a weakly-typed language, some invalid conversions/casts may not be checked (leading to undefined behavior)

Types - A final thought

Type systems empower you to formalize a problem's structure into (user-defined) types. This allows the compiler to verify that structure, enabling you to write more robust software.

Scoping Every language has scoping rules, which govern the visibility of variables and functions within a program. A variable is "in-scope" in a region of a program if it can be explicitly accessed by its name in that region.

```
void foo() {
    int x;
    cout << x; // Just fine
}
void bar() {
```

```
    cout << x; // ERROR! x isn't in bar's scope!
}
```

Scoping rules tell us what variables are visible at every place in the code, and what to do when there are mutiple variables of the same name

Scope of a variable

the range of a program's instructions where the variable is known

```
void foo() {
    int x;
    cout << x;
}
```

"The scope of the x variable is the function foo()."

In-scope we say that a variable is "in-scope" if it can be accessed by its name in a particular part of a program. The x variable is inscope within the foo function because it is defined at the top of the function Scope changes as a program runs! The set of in-scope variables and functions at a particular point in a program is called its lexical environemnt The environment changes as vriables come in or go out of scope

Each variable also has a "lifetime" from its creation to destruction

A variable's lifetime may include times when the variable is in scope, and times when it is not in scope (but still exists and can be accessed indirectly)

Lifetimes...of Values

A variable’s lifetime describes when a variable can be accessed. It includes times when the variable is in scope, but also times where it’s not in scope (but can be accessed indirectly).

Values also have lifetimes. Variables and the values they point to can have different lifetimes!

Take a look at our previous example, but slightly modified:

void study(int how_long) { while (how_long-- > 0) cout << "Study!\n"; cout << "Partay!\n"; }

int main() { int hrs = 10; study(hrs); cout << "I studied " << hrs << " hours!"; }

The scope of hrs is only within main(); when we call into study, we can no longer access it. But, its lifetime also extends to study(), since we can access it indirectly - through the variable how_long!

Values also have lifetimes and they're often independent of variables So while a variable's lifetime is limited to the execution of the function where it's defined...A value may have a lifetime that extends indefinitely

Lexical Scoping (Static)

All programs are comprised of a series of nested contexts: we have files, lasses in those files, functions in those classes, blocks in those functions, blocks within blocks, etc.

With lexical scoping, we determine all variables that are in scope at a position X in our code by looking at X's context first, then looking in successively larger enclosing contexts around x.

Python does scoping using the "LEGB" rule:

Local, Enclosing, Global, Built-in Local: First look in the current code block, function body or lambda expression Enclosing: Then (if you have a nested function) look in the enclosing function that contains your function If a variable/function encloses another variable/function, then the one the encloses can be categorized as enclosing and still within scope inside the "another variable/function". Global: Then look at all of the top-level variables and functions Built-in: Finally you're left with built-in python keywords, functions, etc.

What types of contexts do we consider for Lexical Scope? Expressions: A new variable is introduced as part of an expression, and its scope is limited to that expression Blocks: A new variable is introduced within a block, and its scope is limited to that block. Functions: A local variable or parameter is introduced within a function, and its scope is limited to that function

Classes/Structs: A class can have member variables, whose scope is limited to that class.

Namespaces: Some languages have namespaces that also provide "cleaner" scoping. Global: We can define global variables, whose scope is available to all functions in the program (or file).

Shadowing: code that defines different variables of the same name, but at different scopes.

When a variable is redefined in a different context, lexical scoping languages typically use an approach called shadowing. In an inner context, the redefined variable “replaces” all uses of the outer context variable; once the inner context is finished, we return to the outer context variable.

```
int main(){
  int x = 42;
  int sum = 0;

  for (int i = 0; i < 10; i++) {
    int x = i;
    std::cout << "x: " << x << '\n'; // prints values of i from 0 to 9
    sum += x;
  }

  std::cout << "sum: " << sum << '\n';
  std::cout << "x:   " << x   << '\n'; // prints out 42
}
```

Dynamic Scoping

When you reference a variable, the program tries to find it in the current block and its enclosing blocks... If the variable can't be found, the program then searches the calling function for the variable. If it can't be found there, it checks its calling function Dynamic scoping takes a different approach: the value of a variable is always

the most recently defined (or redefined) version, regardless of the lexical scope! For dynamic scoping, what matters is the chronological order variables are defined in: not how the code is structured.

```
func foo() {
  x = x + 1;
  print x
}

func bar() {
  x = 10;
  foo();
}

func main() {
  x = 1;
  foo();
  bar();
}

// prints:
// 2
// 11
```

Memory Safety:

Memory-Safe languages prevent memory operations that could lead to undefined behaviors. Memory-unsafe languages allow memory operations that could lead to undefined behaviors An inordinate amount of bugs and hacking vulnerabilities are due to memory unsafety.

Memory Unsafe Languages

- allow out-of-bound array indexes and unconstrained pointer arithmetic
- allow casting values to incompatible types
- allow use of uninitialized variables/pointers
- allow use of dangling pointers to dead objects (programmer-controlled object destruction)

Memory Safe languages:

- throw exceptions for out-of-bound array indexes; disallow pointer arithmetic
- throw an exception or generate a compiler error for invalid casts
- throw an exception or generate a compiler error if an uninitialized variable/pointer is used;
- Hide explicit pointers altogether
- Prohibit programmer-controlled object destruction ensure objects are only destroyed when *all* references to them disappear (Garbage collection)

Strategies for Memory Leaks and Dangling Pointers

Garbage Collection: the language manages all memory de-allocation automatically Ownership Model: The compiler ensures pobjects get destroyed when their lifetime ends Garbage Collection is the automatic reclamation of memory whcih was allocated by a proram, but which is no longer referenced. In a language with garbage collection the programmer does not explicitly control object destruction - the lagnauge does. When a

value or object on the heap is no longer referred to, the program evantually detects this at runtime and frees the memory associated with it.

Eliminats Memory Leaks: Ensures memory allocated for objects is freed once it's no longer needed Eliminates Dangling Pointers and Use of Dead Objects: Prevents access to objects after they have been de-allocated Eliminates Double-free Bugs: Eliminates inadvertant attempts to free memory more than once Eliminates Manual Memory Management: Simplifies code by eliminating manual deletion of memory. When should objects be garbage collected? Garbage collect an object when there are no longer any references to that object

Memory Safety and management:

Memory safety is a key property of strongly-typed languages. In short, memory safe languages prevent any memory operations that could lead to undefined behaviour, while memory unsafe languages allow such operations. These include:

- out-of-bound array indexes and unconstrained pointer arithmetic
- casting values to incompatible types
- use of uninitialized variables and pointers
- use of dangling pointers to dead objects (use-after-free, double-free, …)

These are the biggest source of security vulnerabilities in low-level code. In memory safe languages, all the above operations are usually not allowed at compile-time (ex pointer arithmetic or invalid casts) or runtime errors (out-of-bound array indexes).

Do memory leaks lead to undefined behaviour? No! Since memory leaks only lead to unused memory not being freed, they don’t lead to undefined behaviour. Even if the system runs out of memory due to the memory leak, the program is predictably terminated. Therefore, we see no undefined behaviour.

Good things to know

- A Dynamically typed language will have the type checker ensure that each primitive operation is invoked with values of the right types as the program executes)
- A statically typed, the program would not compile! Prior to execution, the type checker would determine the type of every expression (including those like “echo $testOne - $testTwo“ which are never called) and ensure that all operations are compatible with the types of their operands.